Fast Multiple Order-Preserving Matching Algorithms

Myoungji Han[1], Munseong Kang[2], Sukhyeun Cho[2], Geonmo Gu[1], Jeong Seop Sim[2], and Kunsoo Park[1]

[1] Department of Computer Science and Engineering, Seoul National University
{mjhan,gmgu,kpark}@theory.snu.ac.kr
[2] Department of Computer Science and Information Engineering, Inha University
{kmsung1125,csukhyeun}@inha.edu, jssim@inha.ac.kr

Abstract. Given a text T and a pattern P, the order-preserving matching problem is to find all substrings in T which have the same relative orders as P. Order-preserving matching has been an active research area since it was introduced by Kubica et al. [13] and Kim et al. [11]. In this paper we present two algorithms for the multiple order-preserving matching problem, one of which runs in sublinear time on average and the other in linear time on average. Both algorithms run much faster than the previous algorithms.

1 Introduction

Given a text T and a pattern P, the order-preserving matching problem is to find all substrings in T which have the same relative orders as P. For example, given $T = (10, 15, 20, 25, 15, 30, 20, 25, 30, 35)$ and $P = (35, 40, 30, 45, 35)$, P has the same relative orders as the substring $T' = (20, 25, 15, 30, 20)$ of T. In T' (resp. P), the first character 20 (resp. 35) is the second smallest number, the second character 25 (resp. 40) is the third smallest number, the third character 15 (resp. 30) is the smallest number, and so on. See Fig. 1. This problem is naturally generalized to the problem of finding multiple patterns. The order-preserving matching for a single pattern will be called the *single order-preserving matching*, and one for multiple patterns the *multiple order-preserving matching*. In this paper we are concerned with the multiple order-preserving matching problem.

Order-preserving matching was introduced by Kubica et al. [13] and Kim et al. [11], where Kubica et al. [13] defined order relations by order isomorphism of two strings, while Kim et al. [11] defined them explicitly by the sequence of rank values, which they called the *natural representation*. They both proposed $O(n + m \log m)$ time solutions for the single order-preserving matching based on the Knuth-Morris-Pratt algorithm, where n is the length of the text and m is the length of the pattern. Kim et al. [11] also proposed an $O(n \log M)$ time algorithm for the multiple order-preserving matching based on the Aho-Corasick algorithm, where M is the sum of lengths of all the patterns. Henceforth, there has

been considerable research on the single and multiple order-preserving matching problems. For the single order-preserving matching, Cho et al. [4] proposed a method to apply the Boyer-Moore bad character rule to order-preserving matching by using the notion of q-grams. Chhabra and Tarhio [3] presented a more practical solution based on filtering. They first encoded input sequences into binary sequences and then applied standard string matching algorithms as a filtering method. Faro and Külekci [7] improved Chhabra and Tarhio's solution by using new encoding techniques which reduced the false positive rate of the filtering step. For the multiple order-preserving matching, Belazzougui et al. [2] theoretically improved the solution of Kim et al. [11] by replacing the underlying data structure by the van-Emde-Boas tree. They achieved randomized $O(n \cdot \min(\log\log n, \sqrt{\frac{\log r}{\log\log r}}, k))$ time for the search, where r is the length of the longest pattern and k is the number of patterns.

Order-preserving matching has been an active research area and many related problems have been studied such as order-preserving suffix trees [6] and order-preserving matching with k mismatches [8]. Kim et al. [10] extended the representations of order relations from binary relations to ternary relations. With their representations, one can modify earlier order-preserving matching algorithms to accommodate strings with duplicate characters, i.e., a number can appear more than once in a string.

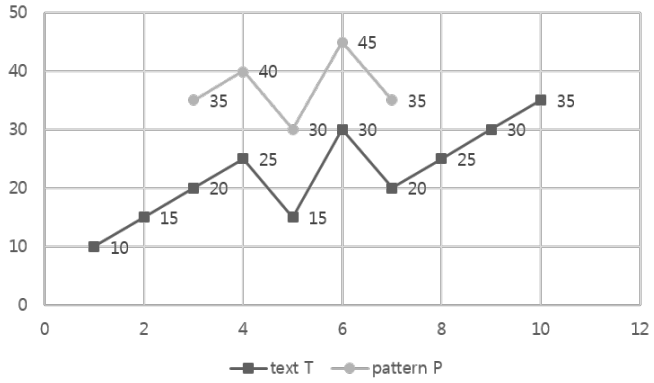

Fig. 1. An example of the order-preserving matching. P has the same relative orders as the substring $T' = (20, 25, 15, 30, 20)$ of T.

In this paper, we present two new algorithms for the multiple order-preserving matching problem which are more efficient on average than the previously proposed algorithms. The algorithms are based on modifications of some conventional pattern matching algorithms such as Wu-Manber [14] and Karp-Rabin [9]. The first algorithm, called Algorithm I, uses the ideas of the Wu-Manber

algorithm, and the second algorithm, called Algorithm II, uses the ideas of the Karp-Rabin algorithm and the encoding techniques of Chhabra and Tarhio [3] and Faro and Külekci [7] for fingerprinting. Algorithm I runs in $O(\frac{n}{m} \log M)$ time on average, where n is the length of the text, m is the length of the shortest pattern, and M is the sum of lengths of all the patterns. Algorithm II runs in $O(n)$ time on average, assuming that M is polynomial with respect to m. In order to verify practical behaviors of our algorithms, we conducted experiments where the two algorithms were compared with the algorithms of Kim et al. [11] and Belazzougui et al. [2]. Experiments show that our algorithms run much faster in practice.

2 Problem Formulation

Let Σ denote a set of numbers such that a comparison of two numbers can be done in constant time, and let Σ^* denote the set of strings over the alphabet Σ. For a string $x \in \Sigma^*$, let $|x|$ denote the length of x. A string x is described by a sequence of characters $(x[1], x[2], ..., x[|x|])$. Let a substring $x[i..j]$ be $(x[i], x[i+1], ..., x[j])$ and a prefix x_i be $x[1..i]$. For a character $c \in \Sigma$, let $rank_x(c) = 1 + |\{i : x[i] < c \text{ for } 1 \leq i \leq |x|\}|$.

We use the *natural representation* defined by Kim et al. [11] to compare order relations of two strings. The natural representation is equivalent to *order-isomorphism* defined by Kubica et al. [13], because the natural representation of two strings are identical if and only if they are order-isomorphic.

Definition 1 (Natural representation [11]). *For a string x of length n, the natural representation is defined as $Nat(x) = (rank_x(x[1]), rank_x(x[2]), ..., rank_x(x[n]))$.*

For example, for $x = (30, 40, 30, 45, 35)$, the natural representation is $Nat(x) = (1, 4, 1, 5, 3)$. We will simply say that x matches y if $|x| = |y|$ and $Nat(x) = Nat(y)$.

Order-preserving matching can be defined in terms of the natural representation.

Definition 2 (Single Order-Preserving Matching [11]). *Given a text $T[1..n] \in \Sigma^*$ and a pattern $P[1..m] \in \Sigma^*$, P matches T at position i if $Nat(P) = Nat(T[i-m+1..i])$. The order-preserving matching is the problem of finding all the positions of T matched with P.*

Definition 2 is naturally generalized to the multiple order-preserving matching, formally defined in Definition 3.

Definition 3 (Multiple Order-Preserving Matching [11]). *Given a text $T[1..n] \in \Sigma^*$ and a set of patterns $\mathcal{P} = \{P_1, P_2, ..., P_k\}$ where $P_i \in \Sigma^*$ for $1 \leq i \leq k$, the multiple order-preserving matching is the problem of finding all the positions of T matched with any pattern in $\mathcal{P}$.*

There are two other representations in addition to the natural representation for comparing order relations of two strings: *prefix representation* and *nearest neighbor representation*. The *prefix representation* can be defined as a sequence of rank values of characters in prefixes.

Definition 4 (Prefix Representation [11]). *For a string x, the prefix representation is defined as $Pre(x) = (rank_{x_1}(x[1]), rank_{x_2}(x[2]), ..., rank_{x_{|x|}}(x[|x|]))$.*

For example, for $x = (30, 40, 30, 45, 35)$, the prefix representation is $Pre(x) = (1, 2, 1, 4, 3)$. We can compute $Pre(x)$ in time $O(|x| \log |x|)$ for general alphabet using the order-statistic tree [11]. The time complexity can be reduced to $O(|x|)$ if the characters can be sorted in $O(|x|)$ time.

Lemma 1. *[4] For two strings x and y where $|x| = |y|$, if x matches y, then $Pre(x) = Pre(y)$.*

The prefix representation has an ambiguity between different strings if they include duplicate characters. For example, when $x = (10, 30, 20)$, and $y = (10, 20, 20)$, the prefix representations of both x and y are $(1, 2, 2)$, whereas their natural representations are different. Kim et al. defined a new representation called the *extended prefix representation* [10] for strings with duplicate characters. We omit the details here.

For the *nearest neighbor representation*, we define $LMax_x[i]$ and $LMin_x[i]$ as follows.

$$LMax_x[i] = \begin{cases} j & \text{if } x[j] = \max\{x[k] : x[k] \le x[i] \text{ for } 1 \le k \le i-1\} \\ -\infty & \text{if no such } j \ , \end{cases}$$

$$LMin_x[i] = \begin{cases} j & \text{if } x[j] = \min\{x[k] : x[k] \ge x[i] \text{ for } 1 \le k \le i-1\} \\ \infty & \text{if no such } j \ . \end{cases}$$

If there are multiple j's for $LMax_x[i]$ or $LMin_x[i]$, we choose the rightmost one.

Definition 5 (Nearest Neighbor Representation [10,11]). *For a string x, the nearest neighbor representation is defined as $NN(x) = \binom{LMax_x[1]}{LMin_x[1]} \binom{LMax_x[2]}{LMin_x[2]} \cdots \binom{LMax_x[|x|]}{LMin_x[|x|]}$.*

For example, for $x = (30, 40, 30, 45, 30)$, the nearest neighbor representation is as follows.

$$\mathrm{NN}(x) = \left(\binom{-\infty}{\infty}, \binom{1}{\infty}, \binom{1}{1}, \binom{2}{\infty}, \binom{3}{3} \right) \ .$$

For convenience, let $x[-\infty] = -\infty$, $x[\infty] = \infty$, $Nat(x)[-\infty] = 0$ and $Nat(x)[\infty] = |x|+1$ for any string x. Then, $Nat(x)[LMax_x[i]] \le Nat(x)[i] \le Nat(x)[LMin_x[i]]$ holds for $1 \le i \le |x|$.

The time complexity for computing $\mathrm{NN}(x)$ is $O(|x| \log |x|)$ [11]. Using this representation, we can check if two strings match in time linear to the size of the input, even when the strings have duplicate characters.

Lemma 2. *[4,10,11,13] Given two strings x and y where $|x| = |y|$, assume $NN(x)$ is computed. Then we can determine whether x matches y in $O(|x|)$ time.*

3 Algorithm I

In this section, we present our first algorithm for the multiple order-preserving matching. Algorithm I is based on the Wu-Manber algorithm, which is widely used for multiple pattern matching. Algorithm I is divided into two steps: the preprocessing step and the searching step.

3.1 Preprocessing Step of Algorithm I

Let m be the length of the shortest pattern, and M be the sum of lengths of all the patterns. We consider only the first m characters of each pattern. Let $\mathcal{P}' = \{P'_1, P'_2, \cdots, P'_k\}$ where $P'_i = P_i[1..m]$ (this notation is provided only for clarity of exposition). In the preprocessing step, we build a SHIFT table and a HASH table based on $\mathcal{P}'$, which are analogous to those of the Wu-Manber algorithm. However, since we are looking for strings matched with patterns in terms of order-preserving matching, we have to consider the order representations of strings rather than strings themselves for comparison. Consider a block of length b on the text, where $b \leq m$. The SHIFT table determines the shift value based on the prefix representation of the given block. Given a block x, we define

$$l_x = \max\{j : Pre(P'_i[j-b+1..j]) = Pre(x) \text{ for } 1 \leq i \leq k, b \leq j \leq m\} .$$

That is, l_x means the position of the rightmost block in any $P'_i \in \mathcal{P}'$ which is likely to match x. Here, the term "is likely to" is used because $Pre(x) = Pre(y)$ does not necessarily mean that x matches y. For convenience, let $l_x = -\infty$ if there is no such block. Then, the SHIFT table is defined as

$$\text{SHIFT}[f(x)] = \min(m - l_x, m - b + 1) ,$$

where $f(x)$ is a fingerprint mapping a block x to an integer used as an index to the SHIFT table. Using the factorial number system [12], we define $f(x)$ as

$$f(x) = \sum_{i=1}^{b} (Pre(x)[i] - 1) \cdot (i-1)! .$$

Note that $f(x)$ maps a block x into a unique integer within the range $[0..b! - 1]$ according to its prefix representation.

Fig. 2-(a) shows the SHIFT table when there are three patterns. Assume that $b = 3$. Consider the block $T[3..5]$. The rightmost block in $\mathcal{P}'$ whose prefix representation equals that of $T[3..5]$ is $P_1[2..4]$. The fingerprint $f(T[3..5])$ is 3. Thus, SHIFT[3] is $m - 4 = 1$. Note that in the figure, we can safely shift the patterns by 1.

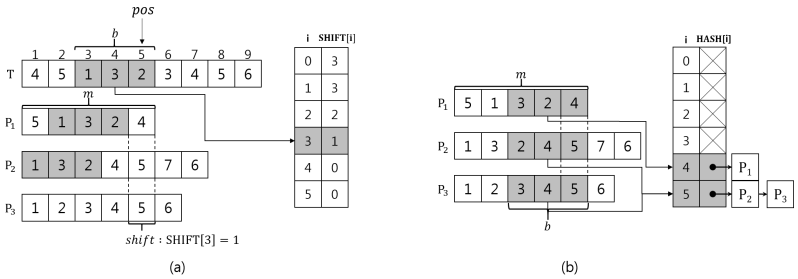

Fig. 2. SHIFT and HASH tables.

The fingerprint is also used to index the HASH table. HASH[i] contains a pointer to the list of the patterns whose last block in $\mathcal{P}'$ is mapped to the fingerprint i. Fig. 2-(b) shows the HASH table with the same patterns.

To compute the values of the SHIFT table, we consider each pattern P'_i separately. For each pattern P'_i, we compute the fingerprint of each block $P'_i[j-b+1..j]$ consecutively, and set the corresponding value of the SHIFT table to the minimum between its current value (initially set to $m-b+1$) and $m-j$. In order to obtain the fingerprint of a block, we have to compute its prefix representation. Once we compute the fingerprint of the first block $Pre(P'_i[1..b])$ using the order-statistic tree, the tree contains the first b characters of P'_i. To compute the prefix representations of the subsequent blocks, we observe that we can compute $Pre(P'_i[j+1..j+b])$ by taking advantage of the order-statistic tree containing characters of the previous block $P'_i[j..j+b-1]$. Specifically, we erase $P'_i[j]$ from the tree and insert the new character $P'_i[j+b]$ into the tree. Inserting and deleting an element into the order-statistic tree is accomplished in $O(\log b)$ time since the tree contains $O(b)$ elements. Then we traverse the tree in $O(b)$ time to retrieve the prefix representation of the new block. We repeat this until we reach the last block. When we reach the last block, we map into the HASH table and add P_i into the corresponding list. The whole process is performed for all the patterns. Since there are $O(km)$ blocks, it takes $O(kmb)$ time to construct the SHIFT and HASH tables.

We also precompute the nearest neighbor representations of all the patterns, namely, $\mathrm{NN}(P_i)$ for $1 \le i \le k$. They are used in the searching step for verifying whether patterns actually match the text. Using the order-statistic tree, they are computed in $O(M \log r)$ time, where r denotes the length of the longest pattern. As a result, the time complexity for the preprocessing step is $O(kmb + M \log r)$.

3.2 Searching Step of Algorithm I

In the searching step, we find all the positions of T matched with any pattern in $\mathcal{P}$. Fig. 3 shows the pseudocode of Algorithm I. For the search, we slide a position pos along the text, reading a block of length b, $T[pos-b+1..pos]$, and computing

the corresponding fingerprint i. If $\text{SHIFT}(i) > 0$, then we shift the search window to $pos + \text{SHIFT}(i)$ and continue the search. Otherwise, $\text{SHIFT}(i) = 0$ and there may be a match. Thus we select the list of patterns in $\text{HASH}[i]$, and compare each pattern in the list with the text via the nearest neighbor representation. We call this process the verification step. We repeat this until we reach the end of the text.

Algorithm I($P = \{P_1, P_2, \cdots, P_k\}$, $T[1..n]$)

```
1: m ← min_{1≤i≤k}(|P_i|)
2: Preprocess P and compute SHIFT, HASH, NN
3: pos ← m
4: while pos ≤ n do
5:     i ← f(T[pos − b + 1..pos])
6:     if SHIFT[i] = 0 then
7:         Verify each pattern in HASH[i] via NN
8:         pos ← pos + 1
9:     else
10:        pos ← pos + SHIFT[i]
11:    end if
12: end while
```

Fig. 3. The pseudocode of Algorithm I

3.3 Average Time for the Search of Algorithm I

We present a simplified analysis of the average running time for the searching step. For the analysis, we assume that there are no duplicate characters in any b-length block in strings, i.e., any consecutive b characters in the text and patterns are distinct. Although this assumption restricts the generality of our problem, it is insignificant because: (1) a fairly large alphabet makes the case against the assumption very unlikely to happen; (2) even if it happens, the algorithm still works correctly without a significant impact on the performance in practice. We leave it as an open problem whether the average $O(\frac{n}{m} \log M)$ time can be derived when the strings are totally random, which is more complicated. Now, we assume that each distinct block appears randomly at a given position (i.e., with the same probability). Let us denote $\sigma = |\Sigma|$, then there are ${}_{\sigma}P_b$ different possible blocks and the probability of a block to appear is $1/{}_{\sigma}P_b$.

Lemma 3. *For two random blocks x and y, where $x, y \in \Sigma^b$ and each has no duplicate characters, the probability that $Pre(x) = Pre(y)$ is $\frac{1}{b!}$.*

Recall that Algorithm I determines a shift value according to the prefix representation of a current block on the text.

Lemma 4. *The probability that a random block x leads to a shift value of j, $0 \leq j \leq m - b$, is at most $\frac{k}{b!}$.*

Lemma 5. *The expected value of a shift during the search is at least* $(m - b + 1)\{1 - \frac{k(m-b+2)}{2b!}\}$.

We set $b = 1.5 \log M / \log \log M$. Then, by Stirling's approximation [1], we can easily prove that $b! = 2^{b \log b + b \log e + O(\log b)} = \Omega(M)$, and thus the expected value of a shift is at least $\Theta(m)$. Consequently, the average number of iterations of the **while** loop during the search is bounded by $O(\frac{n}{m})$. At each iteration, we compute a fingerprint and the computation takes $O(b \log b) = O(\log M)$ time. Lemma 6 shows that the verification step at each iteration is accomplished in constant time on average.

Lemma 6. *The average cost of the verification step at each iteration is* $O(1)$.

Hence, the average time complexity of the searching step is roughly $O(\frac{n}{m} \log M)$.

4 Algorithm II

In this section, we present a simple algorithm that achieves average linear time for search. Algorithm II exploits the ideas of the Karp-Rabin algorithm and the encoding techniques of Chhabra and Tarhio [3] and Faro and Külekci [7] for fingerprinting.

4.1 Fingerprinting in Algorithm II

The Karp-Rabin algorithm is a practical string matching algorithm that makes use of fingerprints to find patterns, and it is important to choose a fingerprint function such that a fingerprint should be efficiently computed and efficiently compared with other fingerprints. Furthermore, the fingerprint function should be suitable for identifying strings in terms of order-preserving matching.

Given an m-length pattern P, Chhabra and Tarhio [3] encode the pattern into a binary sequence $\beta(P)$ of length $m - 1$, where

$$\beta(P)[i] = \begin{cases} 1 & \text{if } P[i] < P[i+1] \\ 0 & \text{otherwise.} \end{cases}$$

We consider the fingerprint $\beta(P)$ as an $(m-1)$-bit binary number. We can compute $\beta(P)$ in time $O(m)$.

As m increases, the fingerprint $\beta(P)$ may be too large to work with; we need at least $(m-1)$ bits to represent a fingerprint. To address this issue, we compute the fingerprints as residues modulo a prime number p. According to [9], we choose the prime p pseudorandomly in the range $[1..mn^2]$. With this choice, it is proved that the probability of a single false positive due to the modulo operation while searching is bounded by $2.53/n$, which is negligibly small for sufficiently large n [9].

Faro and Külekci [7] proposed more advanced encoding techniques such as q-NR and q-NO. Instead of comparing between only a pair of neighboring characters, they compared between a set of q characters for computing the relative

position of a character. We can compute fingerprints using those techniques similarly to above. We implemented Algorithm II using three encoding techniques, including Chhabra and Tarhio's binary encoding [3], q-NR, and q-NO [7], for fingerprinting. In the following sections, we will describe the algorithm assuming the binary encoding.

4.2 Preprocessing Step of Algorithm II

Again, let $\mathcal{P}' = \{P'_1, P'_2, \cdots, P'_k\}$ be the set of m-length prefixes of the patterns. In the preprocessing step, we first compute $\beta(P'_i)$ for $1 \leq i \leq k$ and build a HASH table. HASH[i] contains a pointer to the list of the patterns whose fingerprints equal i. We also compute $\mathrm{NN}(P_i)$ for $1 \leq i \leq k$. In total, the preprocessing step takes $O(M \log r)$ time. Fig. 4-(a) shows the HASH table when there are three patterns. We use a prime $p = 7$ in the example.

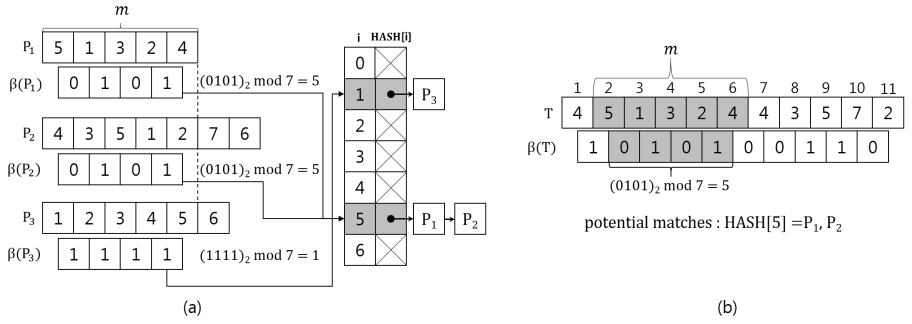

Fig. 4. (a) The HASH table. (b) An example of the search. For the window $T[2..6]$, the corresponding fingerprint is 5. We check HASH[5], which has P_1, P_2 as elements, and thus verify them via NN.

4.3 Searching Step of Algorithm II

In the searching step, we scan the text T while iteratively computing fingerprints of the successive windows of size m. Fig. 5 shows the pseudocode of Algorithm II. We slide a search window $T[i..i+m-1]$ along the text, computing the corresponding fingerprint β. If the list pointed by HASH[β] is not empty, we compare each pattern in the list with the text via its nearest neighbor representation. We call this process the verification step. We repeat this until we reach the end of the text. Fig. 4(b) shows an example of the searching step.

Algorithm II($P = \{P_1, P_2, \cdots, P_k\}$, $T[1..n]$)
1: $m \leftarrow min_{1 \le i \le k}(|P_i|)$
2: Preprocess P and compute HASH, NN
3: $pos \leftarrow m$
4: **for** $i = 1$ for $n - m + 1$ **do**
5: $\beta = \beta(T[i..i + m - 1]) \bmod p$
6: Verify each pattern in HASH[β] via NN
7: **end for**

Fig. 5. The pseudocode of Algorithm II

4.4 Average Time for the Search of Algorithm II

At each iteration of the **for** loop, we compute the fingerprint β of the search window. Let us denote $\beta_i = \beta(T[i..i + m - 1]) \bmod p$, which is the fingerprint of the i-th search window. We can compute β_1 in time $O(m)$. To compute the fingerprints for the subsequent windows, we observe that we can compute β_{i+1} from β_i using Horner's rule [5], since

$$\beta_{i+1} = (2(\beta_i - H \cdot \beta(T)[i]) + \beta(T)[i + m]) \bmod p \,,$$

where $H = 2^{m-2} \pmod p$ is a precomputed value. It is clear that this calculation is done in constant time.

Now, we analyze the time spent to perform the verification step. We assume that the numbers in the text and patterns are statistically independent and uniformly at random. The verification is performed when there is a match between encoded binary strings of the text and patterns. The probability that a 1 appears at a position of an encoded string is $q = (\sigma^2/2 - \sigma/2)/\sigma^2 = (\sigma - 1)/2\sigma$. So the probability of a character match [3] is

$$s = q^2 + (1 - q)^2 = \frac{1}{2} + \frac{1}{2\sigma^2} \,.$$

Since the odd positions of an encoded string are mutually independent, we can (upper) bound the probability of a match between two encoded strings by $s^{(m-1)/2}$. Note that $s \le 5/8$ for $\sigma \ge 2$.

Lemma 7. *When M is polynomial with respect to m, the average cost of the verification step during the search is $O(1)$.*

Hence, the average time complexity of the searching step is $O(n)$, when M is polynomial with respect to m.

5 Experiments

In order to verify the practical behaviors of our algorithms, we tested them against the previous algorithms based on the Aho-Corasick algorithm: Kim et al.'s [11], and Belazzougui et al.'s [2].[1] Kim et al.'s algorithm is denoted by *KEF*,

[1] For the implementation of the van-Emde-Boas tree used in [2], we used the source code publicly available at https://code.google.com/p/libveb/.

Belazzougui et al.'s by *BPR*, and Algorithm I by *Alg1*. Algorithm II is denoted by *Alg2*, followed by a notation of the encoding technique adopted for fingerprinting. Specifically, *Alg2_Bin* refers to Algorithm II with Chhabra and Tarhio's binary encoding [3], and *Alg2_NR2* (resp. *Alg2_NO2*) refers to Algorithm II with the q-NR (resp. q-NO) encoding of Faro and Külekci [7] where we set $q = 2$. All algorithms were implemented in C++ and run on a Debian Linux 7(64bit) with Intel Xeon X5672 processor and 32 GB RAM. During the compilation, we used the O3 optimization option.

We tested for a random text T of length $n = 10^6$ searched for $k = 10, 50, 100$ random patterns of length $m = 5, 10, 20, 50, 100$, respectively. All the texts and patterns were selected randomly from an integer alphabet $\Sigma = \{1, 2, \cdots, 1000\}$ (we tested for varying alphabet sizes, but we didn't observe sensible differences in the results). For each combination of k and m, we randomly selected a text and patterns, and then ran each algorithm. We performed this 10 times and measured the average time for the searching step. Table 1 shows the results.

Fig. 6 in Appendix shows the average search times when $k = 10$. When m is less than 50, *Alg2_Bin* is the best among the algorithms, achieving a speed up of about 6 times compared to *KEF*, and about 14 times compared to *BPR*. As m increases, however, *Alg1* becomes better, achieving a speed up of about 11 times compared to *KEF*, and about 24 times compared to *BPR*. This is due to the increase of the average shift value during the search. The reason that the average shift value increases is that since we set $b = 1.5 \log M / \log \log M$, the block size increases as m increases, and thus the probability that a block appears in the patterns decreases. Fig. 7 and 8 in Appendix show the average search times when $k = 50, 100$. They show similar trends with Fig. 6. One thing to note is that as k increases, the point of m where *Alg1* becomes for the first time faster than the *Alg2* family increases. We attribute this to the fact that as k increases, a block appears more often in the patterns, which leads to lower shift values.

6 Conclusion

We proposed two efficient algorithms for the multiple order-preserving matching problem. Algorithm I is based on the Wu-Manber algorithm, and Algorithm II is based on the Karp-Rabin algorithm and exploits the encoding techniques of the previous works [3,7]. Algorithm I performs the multiple order-preserving matching in average $O(\frac{n}{m} \log M)$ time, and Algorithm II performs it in average $O(n)$ time when M is polynomial with respect to m. The experimental results show that both of the algorithms are much faster than the existing algorithms. When the lengths of the patterns are relatively short, Algorithm II with the binary encoding performs the best due to its inherent simplicity. However, Algorithm I becomes more efficient as the lengths of the patterns grow.

Table 1. Average search times with different values for k and m.

k	m	*KEF*	*BPR*	*Alg1*	*Alg2_Bin*	*Alg2_NR2*	*Alg2_NO2*
10	5	527.3	1215.1	274.8	**107.6**	164.8	186.5
	10	544.3	1258.2	216.9	**91.5**	148.8	197.6
	20	557.1	1254.8	286.5	**88.4**	155.4	194.8
	50	556.2	1213	**51.1**	65.4	116.7	203
	100	561.7	1244.8	**56.2**	70.4	123.9	206.9
50	5	598	1227.2	647.8	234.1	**215.3**	310
	10	573.8	1238.6	269.6	**100.5**	152.3	194.6
	20	562.9	1244.2	308.6	**114.8**	187.1	216.4
	50	570.7	1239.8	313.5	**113.8**	184.5	226.2
	100	587.6	1271.8	**55.4**	86.1	150.6	227.6
100	5	569	1291.1	674.4	395.6	386.3	**307.3**
	10	629	1304.3	522.4	**81.9**	100.3	150.5
	20	589	1250.2	498.4	**102.6**	164	205.1
	50	605.3	1259.9	103.3	**86**	184.8	225.7
	100	588.9	1247.2	73.2	**53.8**	182	227.5

References

1. Abramowitz, M., Stegun, I.A.: In: Handbook of mathematical functions. Dover New York (1972)
2. Belazzougui, D., Pierrot, A., Raffinot, M., Vialette, S.: Single and Multiple Consecutive Permutation Motif Search. In: Cai, L. Cheng, S.-W., Lam, T.-W. (eds.) ISAAC 2013. LNCS, vol.8283, pp. 66–77, Springer, Heidelberg (2013)
3. Chhabra, T., Tarhio, J.: Order-Preserving Matching with Filtration. In: Proc. SEA'14, 13th International Symposium on Experimental Algorithms, pp. 307–314, Springer (2014)
4. Cho, S., Na, J.C., Park, K., Sim, J.S.: A fast algorithm for order-preserving pattern matching. Information Processing Letters 115(2), 397–402 (2015)
5. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: In: Introduction to algorithms 2nd edn. MIT press, Cambridge (2001)
6. Crochemore, M., Iliopoulos, C.S., Kociumaka, T., Kubica, M., Langiu, A., Pissis, S.P., Radoszewski, J., Rytter, W., Waleń, T.: Order-preserving incomplete suffix trees and order-preserving indexes. In: Kurland, O., Lewenstein, M., Porat, E. (eds.) SPIRE 2013. LNCS, vol. 8214, pp. 84–95, Springer, Heidelberg (2013)
7. Faro, S., Külekci, O.: Efficient Algorithms for the Order Preserving Pattern Matching Problem. arXiv preprint arXiv:1501.04001 (2015)
8. Gawrychowski, P., Uznański, P.: Order-preserving pattern matching with k mismatches. In: Combinatorial Pattern Matching, pp. 130–139, Springer (2014)
9. Karp, R.M., Rabin, M.O.: Efficient randomized pattern-matching algorithms. IBM Journal of Research and Development 31(2), 249–260 (1987)
10. Kim, J., Amir, A., Na, J.C., Park, K., Sim, J.S.: On Representations of Ternary Order Relations in Numeric Strings. In: ICABD, pp. 46–52, Springer (2014)
11. Kim, J., Eades, P., Fleischer, R., Hong, S., Iliopoulos, C.S., Park, K., Puglisi, S.J., Tokuyama, T.: Order-preserving matching. Theoretical Computer Science 525, 68–79 (2014)

12. Knuth, D.E.: In: The Art of Computer Programming vol. 2 Seminumerical Algorithms. Addison Wesley (1998)
13. Kubica, M., Kulczyński, T., Radoszewski, J., Rytter, W., Waleń, T.: A linear time algorithm for consecutive permutation pattern matching. Information Processing Letters 113(12), 430-433 (2013)
14. Wu, S., Manber, U.: A fast algorithm for multi-pattern searching. Technical report. TR-94-17, Department of Computer Science, University of Arizona (1994)

Appendix

Proof of Lemma 3. The probability is equivalent to the probability that when randomly choosing a block x, its prefix representation is the same as that of (already chosen) y. The sample space consists of all ${}_{\sigma}P_b$ possible blocks. Notice that once we choose b distinct characters regardless of order, we can order them to fit in any one of the $b!$ prefix representations. It means that there are $\binom{\sigma}{b}$ blocks belonging to each prefix representation. Therefore, the probability is $\binom{\sigma}{b} \cdot \frac{1}{{}_{\sigma}P_b} = \frac{1}{b!}$. □

Proof of Lemma 4. The necessary condition for the case that x leads to a shift value j is that there exists a pattern P'_i whose block ending at the position $m-j$ belongs to the prefix representation of x. Since there are k patterns, the probability of the necessary condition is $\frac{k}{b!}$. □

Proof of Lemma 5. Since all the entries of SHIFT were initialized to $m-b+1$, the expected value of a shift is $\geq \sum_{j=0}^{m-b} j \cdot \frac{k}{b!} + (m-b+1)\{1-(m-b+1)\frac{k}{b!}\} = (m-b+1)\{1-\frac{k(m-b+2)}{2b!}\}$. □

Proof of Lemma 6. At each iteration, the probability that a pattern P_i leads to the verification step is $\frac{1}{b!}$ and the cost for the verification for P_i is $O(|P_i|)$ by Lemma 2. Since there are k patterns, the expected cost of the verification step at each iteration is $\sum_{i=1}^{k} \frac{O(|P_i|)}{b!} = \frac{O(M)}{b!} = O(1)$. □

Proof of Lemma 7. At each iteration, the probability that a pattern P_i leads to the verification is at most $s^{(m-1)/2}$. Thus, the expected cost of the verification step is at most $\sum_{i=1}^{k} s^{(m-1)/2} \cdot O(|P_i|) = O((\frac{5}{8})^{m/2} \cdot M)$, which is $O(1)$ when M is polynomial with respect to m. □

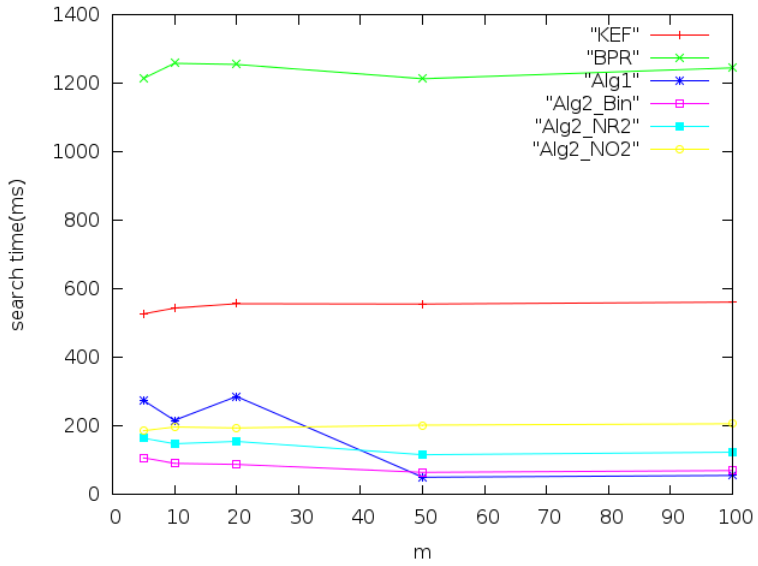

Fig. 6. Average search times when $k = 10$.

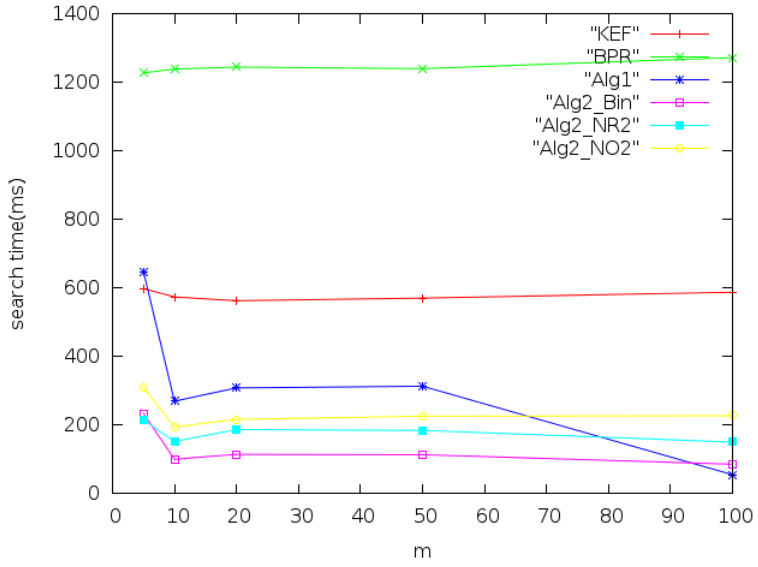

Fig. 7. Average search times when $k = 50$.

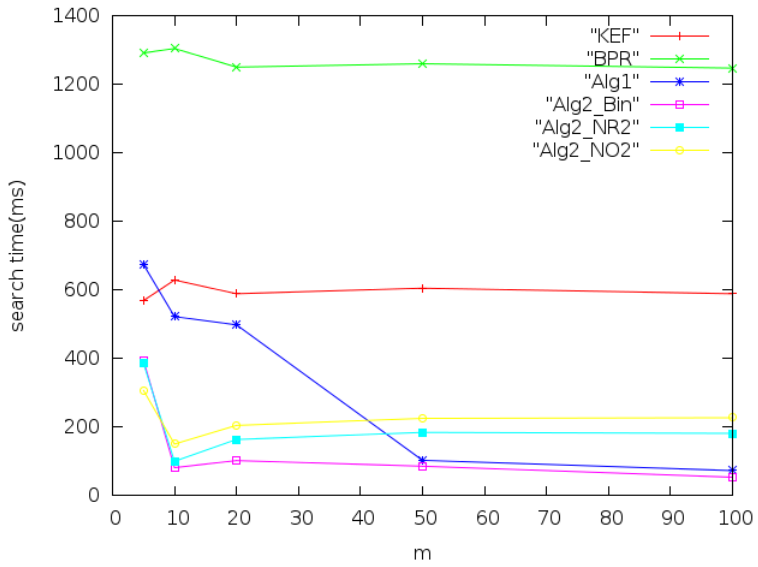

Fig. 8. Average search times when $k = 100$.